Algebraic Derivation of Until Rules and Application to Timer Verification

Jessica Ertel[1], Roland Glück[2], and Bernhard Möller[3]

[1] msg-life (ertel-jessica@hotmail.de)
[2] German Aerospace Center (roland.glueck@dlr.de)
[3] University of Augsburg (moeller@uni-augsburg.de)

Abstract. Using correspondences between linear temporal logic and modal Kleene Algebra, we prove in an algebraic manner rules of linear temporal logic involving the until operator. These can be used to verify programmable logic controllers; as a case study we use a part of the control of pedestrian lights, verified with the interactive tool KIV.

1 Introduction

Overview Semirings, Kleene Algebra and their algebraic relatives have proved to be a flexible tool for reasoning about a broad variety of topics such as graph problems, algorithms and transformations [12,14,23,24,27,30], energy problems [20], fuzzy logic and relations [31], software development and verification [38,41] and database theory [36,40]. Here we use and apply an approach from [17,39] which relates Modal Kleene Algebra (MKA) and Linear Temporal Logic (LTL). It shows a.o. that sets of LTL traces form an MKA and that the standard LTL operators can be represented as compositions of MKA operators. Along these lines we first prove algebraically a few new properties of the LTL *until* operator in MKA. Since we use the MKA formalization, we prove in fact much more general theorems which hold in all MKAs, not just the LTL variant mentioned above. But, of course, the results apply to LTL itself as well.

We apply these in the interactive verification of programmable logic controllers (PLCs). Encouraged by the results in [21] on this, we tackle as a considerably more difficult new task a substantial part of traffic light control systems in PLC, including a formalization of timers. Besides this, the paper deals substantially with temporal phenomena, in which the until operator is a big structuring help. These issues were not yet covered in [21]; the treatment is based on [19]. As verification tool we choose KIV [3], a rather uncommon interactive verifier, hosted at the university of Augsburg. Despite not being widely known, it succeeded in some verification competitions [7,8]. Moreover, the present work continues [21] which also used KIV.

Related Work Recent approaches to PLC verification are simulation based [15], use data-flow analysis [28] or model checking [35,42]. Simulation based approaches and model checking (based on timed automata) in a naïve manner suffer from the same problem: when confronted with a timer they have to execute or check all timer values, see e.g. the handling of the variable `timer` in

the specification `robot.smv` from [4]. There all possible timer values between 0 and 400 are evaluated, whereas only some of them are important for the verification of the system under consideration (note that this is not a property of the complexity of the system's description and formalization but rather of the employed verification mechanism). Verification of timed PLC programs is a very sparse field; e.g., [35] deals only with Boolean values, whereas [42] explicitely excludes timers. An interactive approach to PLC timer verification using COQ [1] is presented in [47]. This is closely related to our approach; however, it does not reason about until-properties as we will do here.

Our Contribution The present paper introduces an interactive approach to the verification of timed programs, in particular to timed PLC programs. Based on an algebraic modeling and a timer formalization that takes only significant values of the timer into account, we circumvent the problems sketched above. This idea can be seen as a variant of the zone-graph technique (see [9,48]) which divides the set of possible clock values into equivalence classes and achieves a model considering only important clock values. Formalization in MKA and proofs were conducted in KIV [3] due to the preliminary work [21] and our desire to show the possibility of a purely algebraic approach. As a side effect, interactive verification has the potential to guide humans to better bug-fixing than model checking which only outputs faulty traces. Moreover, algebraic rules as derived in Section 2.3 can be deployed in a large context not restricted to a single particular formalization. Finally, algebraic reasoning is much more compact and needs much fewer steps than pointwise reasoning in the original formulation of LTL or Dynamic Logic, even though the latter is directly supported by KIV.

Structure The paper is organized as follows: In Sections 2.1 and 2.2 we recall the basics of MKA and the connection between MKA and LTL. Section 2.3 gives algebraic proofs of some important rules concerning the until operator. Section 3 adapts and substantially extends the earlier results on PLC verification from [21]. After a quick introduction to PLCs in Section 3.1 and their MKA modeling in Section 3.2, we show in Section 3.3 how to formalize a PLC timer in our framework. Section 3.4 ties all threads together in a case study verifying a central part of the control of pedestrian lights. Conclusion and outlook are given in Section 4.

2 Modal Kleene Algebra and Linear Temporal Logic

For this section we assume basic knowledge about lattice theory and semirings (e.g. [13,22,29]), and about temporal logic (e.g. [11,34]). As usual, we often omit the semiring multiplication sign for better readability. Furthermore, we use $\sum$ and $\prod$ for general finite sums and products in semirings.

2.1 Modal Kleene Algebra

As stated in Sect. 1, MKA is a by now well established subdiscipline of Algebraic Logic with numerous applications. Its several axiomatic variants have different advantages and disadvantages; hence we make precise which one we use.

First, a *Kleene algebra* [32] is a structure $(M, +, \cdot, 0, 1, ^*)$ where $(M, +, \cdot, 0, 1)$ is an idempotent semiring with *natural order* $x \leq y \Leftrightarrow x + y = y$, and the *Kleene star* operator * satisfies the following axioms for all $x, y, z \in M$:

$$\begin{array}{lll} 1 + xx^* \leq x^* & 1 + x^*x \leq x^* & \text{(right and left unfold)} \\ y + zx \leq z \Rightarrow yx^* \leq z & y + xz \leq z \Rightarrow x^*y \leq z & \text{(right and left induction)} \end{array}$$

Star has many useful properties like *reflexivity*, *multiplicative idempotence* and *isotony*, i.e., $1 \leq x^*$, $x^* \cdot x^* = x^*$ and $x \leq y \Rightarrow x^* \leq y^*$ for all x, y.

Assume now an idempotent semiring $S = (M, +, \cdot, 0, 1)$ whose elements correspond to sets of possible transitions (e.g., relations) between states of some kind. In particular, 0 models the empty set of transitions. To get an algebraic representation for sets of states one introduces the notion of tests [37,26,33]. An element $p \in M$ is called a *test* if there exists an element $\neg p$ (the *complement* of p) such that $p + \neg p = 1$ and $p \cdot \neg p = 0 = \neg p \cdot p$ hold. Clearly, all tests p satisfy $p \leq 1$. In the case of relations, tests are subrelations of the identity relation and hence can indeed be viewed as representations of sets of states. The set $\mathsf{test}(S)$ of all tests of S forms a Boolean algebra with multiplication as infimum, addition as supremum and $\neg$ as complement operator. Moreover, 0 and 1 are the least and greatest tests, with 0 also representing the empty set of states. For that reason, in view of the formal semantics of LTL to come, we call a test p *valid*, in signs $\models p$, if $p = 1$ (equivalently, if $1 \leq p$). Finally, it is useful to define test implication by $p \rightarrow q =_{df} \neg p + q$. This satisfies the important *shunting* equivalence $p \cdot q \leq r \Leftrightarrow p \leq q \rightarrow r$, with $p \cdot q \leq r \Leftrightarrow p \leq \neg q + r$ as a consequence. Moreover, this implies $\models p \rightarrow q \Leftrightarrow p \leq q$.

In an idempotent semiring $S = (M, +, \cdot, 0, 1)$ one can axiomatize the *(forward) diamond* operator $|\ \rangle$ of type $M \times \mathsf{test}(S) \rightarrow \mathsf{test}(S)$ by the equivalence $|x\rangle p \leq q \Leftrightarrow_{df} \neg qxp \leq 0$ for all $x \in M$ and $p, q \in \mathsf{test}(S)$. In the case of existence, the operator is unique. The test $|x\rangle p$ represents the inverse image of p under x, i.e., the states that are related by x to at least one p-state. A backward diamond $\langle\ |$ representing the image operator can be defined symmetrically by $\langle x|p \leq q \Leftrightarrow_{df} px\neg q \leq 0$ for all $x \in M$ and $p, q \in \mathsf{test}(S)$.

The diamonds distribute over $+$ and hence are isotone in both arguments. Moreover, the *import/export law* $|px\rangle q = p(|x\rangle q)$ and its dual hold for all x and tests p, q. Finally, the diamonds of tests are characterized by $|p\rangle q = pq = \langle p|q$ (and hence, in particular, $|1\rangle q = q = \langle 1|q)$ for all $p, q, \in \mathsf{test}(S)$.

The structure $(M, +, \cdot, 0, 1, |\ \rangle, \langle\ |)$ is called a *modal semiring* if additionally the *modality condition* $|xy\rangle p = |x\rangle|y\rangle p$ and its dual hold for all x, y and tests p.

As the De Morgan dual of the diamonds we introduce the *boxes* by the equality $|x]p =_{df} \neg|x\rangle\neg p$ and its dual. These operators are isotone in their second but antitone in their first argument. Clearly, in a modal semiring we also have $|xy]p = |x]|y]p$ and its dual.

Finally, we call a structure $(M, +, \cdot, 0, 1, ^*, |\ \rangle, \langle\ |)$ a *Modal Kleene Algebra* (or briefly *MKA*) if $(M, +, \cdot, 0, 1, ^*)$ is a Kleene Algebra and $(M, +, \cdot, 0, 1, |\ \rangle, \langle\ |)$ forms a modal semiring. Every MKA satisfies the important *modal star unfold*

and induction rules (and their right duals) for all x and tests p, q:

$$\begin{array}{ll} p + |x\rangle|x^*\rangle p \leq |x^*\rangle p \ , & |x^*]p \leq p \cdot |x]|x^*]p \ , \\ q \leq p \wedge |x\rangle p \leq p \Rightarrow |x^*\rangle q \leq p \ , & p \leq q \wedge p \leq |x]p \Rightarrow p \leq |x^*]q \ . \end{array} \tag{1}$$

MKAs are related to Dynamic Algebras (e.g. [43,25]); a decisive difference is that, via tests, MKAs allow nested modalities such as $|a \cdot |b\rangle p]q$ which restricts a to target states in an inverse image under a transition b. The relationship between Dynamic Algebras, MKAs and Test Algebras has been worked out in [18]. Variants of the modal operators are also present in [10] and the algebraic counterpart [45], which is a special case of MKA. But no temporal operators are treated there. Finally, we mention the framework in [46]; this is rather specialised, whereas we are interested in re-using the more general framework of MKAs. The details are not relevant to the present paper and hence omitted.

2.2 Modal Kleene Algebra and Linear Temporal Logic

The syntax of the language Ψ of LTL formulas over a set Φ of atomic propositions is given by the context-free grammar

$$\Psi \ ::= \ \bot \mid \Phi \mid \neg\Psi \mid \Psi \to \Psi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \mathsf{O}\,\Psi \mid \Box\,\Psi \mid \Diamond\,\Psi \mid \Psi \,\mathsf{U}\, \Psi$$

where $\bot$ denotes falsity, $\to$ is logical implication and O and U are the *next-time* and *until* operators. We are well aware of the redundancies in this definition; they serve to make the presentation of the semantics smoother.

In [39] (refined in [16]) a correspondence between MKA and LTL was established. It uses an MKA $S = (M, +, \cdot, 0, 1, ^*, |\ \rangle, \langle\ |)$ and an element $a \in M$ that models a transition relation transforming a set of states into the set of their successors. Then to every LTL formula ψ one assigns as semantics a test $[\![\psi]\!] \in \mathsf{test}(S)$ that represents the states in which ψ holds. Strictly speaking, the semantic function should be parametrised with the transition element a in the form $[\![\psi]\!]_a$; we omit this for better readability.[4] Further explanations can be found in [39,16]. Since the algebraic semantics only uses the forward diamond and box, we omit the word "forward" in the sequel.

We assume that to every atomic proposition $\varphi \in \Phi$ a test $[\![\varphi]\!] \in \mathsf{test}(S)$ has been assigned as the semantics. Then the semantics of the remaining formulas is inductively defined as follows.

$$\begin{array}{lll} [\![\bot]\!] & = 0 & [\![\mathsf{O}\psi]\!] = |a\rangle[\![\psi]\!] \\ [\![\neg\psi]\!] & = \neg[\![\psi]\!] & [\![\psi_1 \,\mathsf{U}\, \psi_2]\!] = |([\![\psi_1]\!] \cdot a)^*\rangle[\![\psi_2]\!] \\ [\![\psi_1 \to \psi_2]\!] & = [\![\psi_1]\!] \to [\![\psi_2]\!] & [\![\Diamond\,\psi]\!] = |a^*\rangle[\![\psi]\!] \\ [\![\psi_1 \wedge \psi_2]\!] & = [\![\psi_1]\!] \cdot [\![\psi_2]\!] & [\![\Box\,\psi]\!] = |a^*][\![\psi]\!] \\ [\![\psi_1 \vee \psi_2]\!] & = [\![\psi_1]\!] + [\![\psi_2]\!] & \end{array}$$

The semantics of U can be understood as follows. The element $[\![\psi_1]\!] \cdot a$ models the restriction of the transition relation a to those starting states that satisfy ψ_1.

[4] This abstracts from the classical LTL semantics in terms of sets of infinite traces of program states. That concrete semantics is mirrored by a modal semiring in which the elements are relations between sets of traces and tests are sets of traces; states in the sense of the above wording are then single traces, not program states.

Thus, a transition along $(\llbracket\psi_1\rrbracket\cdot a)^*$ traverses only ψ_1-states. Hence $|(\llbracket\psi_1\rrbracket\cdot a)^*\rangle\llbracket\psi_2\rrbracket$ characterizes those states from which ψ_2-states can be reached by traversing only ψ_1-states; this is a faithful representation of the informal U-semantics. Finally, $\Diamond\,\psi$ and $\Box\,\psi$ hold if ψ holds in some/all subsequent states.

Motivated by these definitions we introduce temporal operators on tests by

$$\mathsf{O}p =_{df} |a\rangle p \quad p\,\mathsf{U}\,q =_{df} |(p\cdot a)^*\rangle q \quad \Diamond\, q =_{df} |a^*\rangle q \quad \Box\, q =_{df} |a^*]q \qquad (2)$$

for transition element a and tests p, q. This allows LTL formulas for tests.

Formula ψ_1 *entails* formula ψ_2, in signs $\psi_1 \models \psi_2$, if $\llbracket\psi_1\rrbracket \leq \llbracket\psi_2\rrbracket$. Formula ψ is *valid*, in signs $\models \psi$, if $\top \models \psi$, where $\top = \neg\bot$ is the true formula with $\llbracket\top\rrbracket = 1$. Since 1 is the greatest test, this is equivalent to $\llbracket\psi\rrbracket = 1$. We provide a frequently used rule concerning the validity of $\rightarrow$-formulas: by the above remark, the semantics of $\rightarrow$ and the shunting equivalence we obtain

$$\models \psi_1\rightarrow\psi_2 \Leftrightarrow 1 \leq \llbracket\psi_1\rightarrow\psi_2\rrbracket \Leftrightarrow 1 \leq \llbracket\psi_1\rrbracket\rightarrow\llbracket\psi_2\rrbracket \Leftrightarrow \llbracket\psi_1\rrbracket \leq \llbracket\psi_2\rrbracket \Leftrightarrow \psi_1 \models \psi_2 \quad (3)$$

Given a transition system, the relation transforming a set of states into the set of their successor states is a total function from sets to sets. In MKA, this behavior of an abstract relation a can be enforced by the requirement $|a\rangle p = |a]p$ for all tests p [16,39]; we call an element with this property also a *total function*.

Using the above correspondences, we can prove rules from LTL in an algebraic way, which avoids reasoning about traces and single states.

As an example, we show $\models \psi \rightarrow \Diamond\,\psi$: by (3), the semantics of $\Diamond$ and isotony of the diamond together with $|1\rangle p = p$, we obtain

$$\models \psi \rightarrow \Diamond\,\psi \;\Leftrightarrow\; \llbracket\psi\rrbracket \leq \llbracket\Diamond\,\psi\rrbracket \;\Leftrightarrow\; \llbracket\psi\rrbracket \leq |a^*\rangle\llbracket\psi\rrbracket \;\Leftarrow\; 1 \leq a^* \;,$$

which holds by the definition of star.

2.3 Investigating the Until Operator

In this section we show some useful properties of the LTL until operator using the correspondences with MKA from the previous subsection. All proofs were also done interactively with the KIV system (see [3]), based on the work from [21]. The whole KIV treatment can be found online at [6]; however, we include the proofs to give the reader an impression of the algebraic framework and to demonstrate the power of reasoning in MKA. KIV has also the ability to conduct automated reasoning using adjustable heuristics. Since formulating these is not an easy task, we mostly forwent this feature; exploring its power in our setting will be future work. However, our experience so far shows that the right adjustment of heuristics can help a lot.

It turns out that many proofs about the transition element a only need the weaker condition $|a\rangle p \leq |a]p$ for all tests p. Such an element is called *modally deterministic*. We will show a number of properties of U over such elements.

First, assume that φ implies $\Diamond\,\psi$ and that for every state satisfying φ the (by determinacy unique) successor state satisfies $\varphi \vee \psi$. We will prove that then φ implies $\varphi\,\mathsf{U}\,\psi$. In LTL notation, if $\models \varphi \rightarrow \mathsf{O}(\varphi \vee \psi)$ and $\models (\varphi \rightarrow \Diamond\,\psi)$ then $\models \varphi \rightarrow \varphi\,\mathsf{U}\,\psi$ [5]. To save notation, in the sequel we identify formulas with their

[5] Note that this is not the same as $((\varphi \rightarrow \mathsf{O}(\varphi \vee \psi)) \wedge (\varphi \rightarrow \Diamond\,\psi)) \models \varphi \rightarrow \varphi\,\mathsf{U}\,\psi$, which does not hold.

semantic values. E.g., p, q will stand for the values $[\![\varphi]\!], [\![\psi]\!]$ of formulas φ, ψ. With this convention, a translation into MKA looks as follows (remember (2) and the correspondence of $\wedge/\vee$ with $\cdot/+$):

Lemma 1. *For a modally deterministic element a and tests p, q, if $\models p \to \Diamond q$ and $\models p \to \circ(p + q)$ then $\models p \to (p \,\mathsf{U}\, q)$.*

Proof. Plugging in the definitions and using (3) transform the claim into $p \leq |a\rangle(p + q) \;\wedge\; p \leq |a^*\rangle q \Rightarrow p \leq |(p \cdot a)^*\rangle q$.
First, by idempotence of multiplication on tests and the second assumption $p \leq |a^*\rangle q$ we obtain $p = p \cdot p \leq p \cdot |a^*\rangle q$. So we are done if we can show $p \cdot |a^*\rangle q \leq |(p \cdot a)^*\rangle q$. By shunting and diamond star induction, introducing $r =_{df} \neg p + |(p \cdot a)^*\rangle q$, we obtain

$$p \cdot |a^*\rangle q \leq |(p \cdot a)^*\rangle q \;\Leftrightarrow\; |a^*\rangle q \leq \neg p + |(p \cdot a)^*\rangle q \;\Leftarrow\; q \leq r \wedge |a\rangle r \leq r$$

The first conjunct of the latter fomula holds by $1 \leq (p \cdot a)^*$ and hence $q \leq |(p \cdot a)^*\rangle q \leq r$. For the second one we continue as follows:

$$\begin{array}{ll}
& |a\rangle r \leq r \\
\Leftrightarrow & p \cdot |a\rangle r \leq |(p \cdot a)^*\rangle q \qquad \{\!\![\text{ definition of } r \text{ and shunting back }]\!\!\} \\
\Leftrightarrow & p \cdot |a\rangle \neg p \leq s \wedge p \cdot |a\rangle s \leq s \qquad \{\!\![\text{ setting } s =_{df} |(p \cdot a)^*\rangle q, \text{ definition of } r, \\
& \qquad\qquad \text{distributivity of } |a\rangle \text{ and } \cdot, \text{ lattice algebra }]\!\!\}
\end{array}$$

For the second conjunct we reason as follows:

$$\begin{array}{ll}
& p \cdot |a\rangle s \\
= & |p \cdot a\rangle s \qquad \{\!\![\text{ import/export }]\!\!\} \\
= & |p \cdot a\rangle |(p \cdot a)^*\rangle q \qquad \{\!\![\text{ definition of } s \,]\!\!\} \\
= & |p \cdot a \cdot (p \cdot a)^*\rangle q \qquad \{\!\![\text{ modality }]\!\!\} \\
\leq & |(p \cdot a)^*\rangle q \qquad \{\!\![\, xx^* \leq x^* \text{ by right star unfold, isotony of} \\
& \qquad\qquad \text{diamond }]\!\!\} \\
= & s \qquad \{\!\![\text{ definition of } s \,]\!\!\}
\end{array}$$

The first conjunct is the place where the first assumption is used:

$$\begin{array}{ll}
& p \leq |a\rangle(p + q) \\
\Leftrightarrow & p \leq |a\rangle p + |a\rangle q \qquad \{\!\![\text{ distributivity }]\!\!\} \\
\Leftrightarrow & p \cdot \neg |a\rangle p \leq |a\rangle q \qquad \{\!\![\text{ shunting }]\!\!\} \\
\Leftrightarrow & p \cdot |a]\neg p \leq |a\rangle q \qquad \{\!\![\text{ definition forward box,} \\
& \qquad\qquad \text{Boolean algebra }]\!\!\} \\
\Rightarrow & p \cdot |a\rangle \neg p \leq |a\rangle q \qquad \{\!\![\text{ modal determinacy of } a \,]\!\!\} \\
\Rightarrow & p \cdot p \cdot |a\rangle \neg p \leq p \cdot |a\rangle q \qquad \{\!\![\text{ isotony }]\!\!\} \\
\Leftrightarrow & p \cdot |a\rangle \neg p \leq |p \cdot a\rangle q \qquad \{\!\![\text{ idempotence of test mul-} \\
& \qquad\qquad \text{tiplication, import/export }]\!\!\} \\
\Rightarrow & p \cdot |a\rangle \neg p \leq |(p \cdot a)^*\rangle q \qquad \{\!\![\text{ star unfold and isotony }]\!\!\}
\end{array}$$

□

Next we relate U with $\Box$.

Lemma 2. *Assume an MKA, a modally deterministic element a and tests p, q. Set $u =_{df} q \,\mathsf{U}\, p$ and assume $\models p \to \circ u$.*
i) $\models p \to \Box u$.

ii) If additionally $\models p \to q$ *then* $\models p \to \Box\, q$.

Proof. i) The claim transforms into $p \leq |a^*]u$. So we are done if we can show $p \leq u$ and $u \leq |a^*]u$. The first conjunct holds by diamond star unfold (1). The second conjunct reduces by box star induction (1) to $u \leq u \wedge u \leq |a]u$, of which the first part holds trivially. The second part is, by determinacy of a, implied by $u \leq |a\rangle u$. To show that we calculate, using the definition of u with diamond star unfold, $q \leq 1$ with isotony of diamond and the assumption, $u = p + |q \cdot a\rangle u \leq p + |a\rangle u = |a\rangle u$.

ii) This follows from Part i) by isotony of box if we can show $u \leq q$. To this purpose, we reason as follows:

$$
\begin{array}{lll}
& u \leq q & \\
\Leftrightarrow & |(q \cdot a)^*\rangle p \leq q & \{\!\![\text{ definition of } u \,]\!\!\} \\
\Leftarrow & p \leq q \wedge |q \cdot a\rangle q \leq q & \{\!\![\text{ diamond star induction }]\!\!\} \\
\Leftrightarrow & \mathsf{TRUE} \wedge q \cdot |a\rangle q \leq q & \{\!\![\text{ assumption, import/export }]\!\!\} \\
\Leftrightarrow & \mathsf{TRUE} & \{\!\![\, |a\rangle q \leq 1, \text{ isotony }]\!\!\}
\end{array}
$$

□

The next lemma shows that $\models p \wedge \neg q \to \circ p$ implies $\models p \wedge \Diamond\, q \to p \,\mathsf{U}\, q$ and $\models (r \to \circ(p \wedge \Diamond\, q)) \wedge (p \wedge \neg q \to \circ p)$ implies $\models r \to \circ(p \,\mathsf{U}\, q)$.

Lemma 3. *In an MKA we have for all total functions* a *and tests* p, q, r *the following properties:*

i) $p \cdot \neg q \leq |a\rangle p \;\Rightarrow\; p \cdot |a^*\rangle q \leq |(pa)^*\rangle q$

ii) $r \leq |a\rangle(p|a^*\rangle q) \;\wedge\; p \cdot \neg q \leq |a\rangle p \;\Rightarrow\; r \leq |a\rangle(|(pa)^*\rangle q)$

Proof. i) We reason as follows:

$$
\begin{array}{lll}
& p|a^*\rangle q \leq |(pa)^*\rangle q & \\
\Leftrightarrow & |a^*\rangle q \leq \neg p + |(pa)^*\rangle q & \{\!\![\text{ shunting }]\!\!\} \\
\Leftarrow & q + |a\rangle(\neg p + |(pa)^*\rangle q) \leq \neg p + |(pa)^*\rangle q & \{\!\![\text{ diamond induction }]\!\!\} \\
\Leftrightarrow & q \leq \neg p + |(pa)^*\rangle q \wedge & \{\!\![\text{ lattice algebra }]\!\!\} \\
& |a\rangle(\neg p + |(pa)^*\rangle q) \leq \neg p + |(pa)^*\rangle q &
\end{array}
$$

The first conjunct is shown easily: $q \leq |(pa)^*\rangle q$ holds due to $1 \leq (pa)^*$ and isotony of $|\,\rangle$. Now adding $\neg p$ cannot decrease the right hand side.

For the second conjunct we argue first as follows:

$$
\begin{array}{lll}
& |a\rangle(\neg p + |(pa)^*\rangle q) \leq \neg p + |(pa)^*\rangle q & \\
\Leftrightarrow & p \cdot |a\rangle(\neg p + |(pa)^*\rangle q) \leq |(pa)^*\rangle q & \{\!\![\text{ shunting }]\!\!\} \\
\Leftrightarrow & p \cdot |a\rangle \neg p + p \cdot |a\rangle(|(pa)^*\rangle q) \leq |(pa)^*\rangle q & \{\!\![\text{ distributivity }]\!\!\} \\
\Leftrightarrow & p \cdot |a\rangle \neg p \leq |(pa)^*\rangle q \wedge & \{\!\![\text{ sum properties }]\!\!\} \\
& p \cdot |a\rangle(|(pa)^*\rangle q) \leq |(pa)^*\rangle q &
\end{array}
$$

A shunted form of the second conjunct was already shown in the proof of Lemma 1. The first one follows from the assumption $p \cdot \neg q \leq |a\rangle p$ as follows:

$$
\begin{array}{lll}
& p \cdot \neg q \leq |a\rangle p & \\
\Leftrightarrow & p \cdot \neg |a\rangle p \leq q & \{\!\![\text{ shunting, twice }]\!\!\} \\
\Rightarrow & p \cdot \neg |a\rangle p \leq |(pa)^*\rangle q & \{\!\![\, 1 \leq x^*, \text{ diamond properties }]\!\!\} \\
\Leftrightarrow & p \cdot |a\rangle \neg p \leq |(pa)^*\rangle q & \{\!\![\text{ definition of box and } a \text{ being total and deterministic }]\!\!\}
\end{array}
$$

ii) By Part i) the second conjunct of the premiss of Part ii) implies $p \cdot |a^*\rangle q \leq |(pa)^*\rangle q$ by Part i). Isotony of $|\ \rangle$ yields $|a\rangle(p \cdot |a^*\rangle q) \leq |a\rangle(|(pa)^*\rangle q)$, and now the assumption $r \leq |a\rangle(p|a^*\rangle q)$ and transitivity of $\leq$ show the claim. □

3 Verifying Programmable Logic Controllers

We now apply our semantic foundations to a concrete verification task.

3.1 Basics of Programmable Logic Controllers

Programmable logic controllers (PLCs) are widely used for the control of robots, plants and mechanical devices. They work in a cyclic way: in each cycle they read values from inputs (which stem from the environment and may be, e.g., switch signals or sensor values) and internal variables (which serve for storing values during the execution); from these they compute new values of the internal and output variables (which are forwarded to the environment and may, e.g., start or stop a machine or control the speed of a motor). By default, the names of input and output variables start with `IN` and `OUT`, resp., whereas internal variables have the form `M`x or `M`$x.y$; here the latter form is used to access single bits. It is possible to use variable aliasing to improve readability. Standards for PLCs are defined in [2]; we follow closely the syntax of STEP7 (see [5]).

One of the most common notations for PLCs is provided by function block diagrams (FBD) which use rectangles to represent predefined functions, such as elementary Boolean gates. The inputs of such a rectangle or *block* are on its left side, the outputs on its right. For instance, a block corresponding to conjunction has an ampersand (`&`) at its top, whereas a disjunction is symbolized by `>=1`. The negation of an input or output variable is denoted by a small circle. Normally, no block can ever change the value of an input from the environment; but see Section 3.2 for an exception.

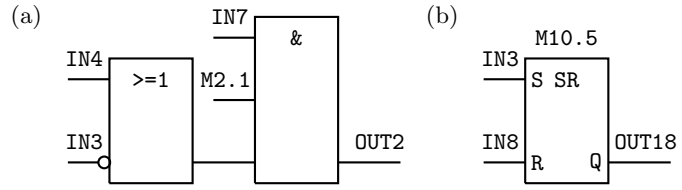

Fig. 1. Boolean Functions (a) and an SR-Flip-flop (b) in FBD

More complex functions can be obtained by linking elementary rectangles, where the *evaluation order* is from left to right and from top to bottom. So the FBD in Figure 1(a) computes the Boolean function (`IN4` $\vee$ $\neg$`IN3`) $\wedge$ `IN7` $\wedge$ `M2.1` and returns the result on output `OUT2`.

Blocks for logical connectives lack the possibility of dynamic behavior and storing of values. A *flip-flop* is an elementary block with such abilities. Flip-flops have two inputs: one set and one reset input, marked by `S` and `R` in their FBDs.

Moreover, they have an internal variable (called *marker*, in FBDs written above the top line) and an output Q which always has the same value as the marker. If the set input is TRUE and the reset input is FALSE then output and marker are set to TRUE. A FALSE-signal on the set input and a TRUE-signal on the reset input set FALSE output and marker to FALSE. If both the set and reset inputs receive a FALSE-signal then the values of output and marker remain unchanged. A set/reset conflict occurs if both the set and reset inputs are TRUE. There are two types of flip-flops, namely set-dominant and reset-dominant or RS- and SR-flip-flops, resp. Upon a set/reset conflict, an RS-flip-flop sets marker and output to FALSE, while an SR-flip-flop sets both to TRUE. Figure 1(b) shows the FBD of a reset-dominant flip-flop with IN3 on its set input, IN8 on its reset input, and output and internal markers OUT18 and M10.5 (which by the above conventions refers to bit 5 of variable M10), resp.

As the last elementary block we consider a simple assignment as shown in Figure 2(a). It assigns the value of IN5 to the internal variable M20.3. Such blocks can be used for every data type (of course, the variables involved have to be of compatible types, as in other programming languages).

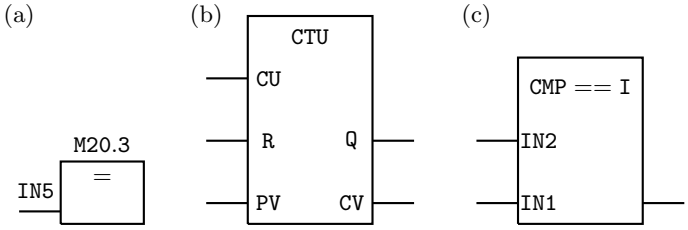

Fig. 2. Assignment, Counter and Comparator in FBD

Although our formalization mostly works with Boolean values, we also need some blocks working with non-Boolean values. For instance, an important further concept is that of a counter, depicted in Figure 2(b). It has two Boolean inputs CU and R, one integer input PV, one Boolean output Q and one integer output CV. A TRUE-signal on R resets the counter to zero. The counter value of the output CV is increased by one upon a positive edge (i.e., a change from FALSE to TRUE) on the input CU. The input PV can be used for setting the counter to a desired value (so one can also reset the counter by feeding zero into it). Finally, the output Q returns the truth value of the comparison $\mathtt{CV} \neq \mathtt{0}$.

Almost self-explanatory, the FBD of Figure 2(c) is a comparator which compares the numerical values of its inputs IN1 and IN2.

In order to obtain timed signals most PLCs offer the possibility of configuring the single bits of a specified internal byte as pulse generators with various frequencies. Often one chooses the byte M100 and assigns to the single bits frequencies as in Table 3. In the sequel, we will follow this convention.

The signal corresponding to one such bit is a wave of rectangular pulses with the associated frequency. E.g., the signal corresponding to bit 100.5 is set alternately half a second to TRUE and half a second to FALSE.

Bit	M100.7	M100.6	M100.5	M100.4	M100.3	M100.2	M100.1	M100.0
Frequency	2 Hz	1.6 Hz	1 Hz	0.8 Hz	0.5 Hz	0.4 Hz	0.2 Hz	0.1 Hz

Fig. 3. Common Frequencies of Pulse Generators

3.2 Modeling Function Block Diagrams in Modal Kleene Algebra

As already shown in [21], FBD programs can be translated into MKA expressions modeling their behavior. The representation uses a glassbox view of components, i.e., all connections and their names are visible. In a relational model then a state is a function from the set of all names to values, and a component (and hence even the whole program) a relation between states. Since by the PLC conventions evaluation follows the left-to-right top-to-bottom diagram order, one can describe a composite component as a linear sequence of relational compositions of elementary components. However, one can abstract from the relational view by associating with each elementary block an MKA element and considering as components only linear products of such elements.

Boolean connections and the values of the corresponding input, output and internal variables can be modeled by tests. However, following PLC conventions, each Boolean is represented by a pair of values with the coupling invariant that they always carry complementary values. Hence an abstract variable $\mathtt{v}$ is represented by the pair $(\mathtt{v}_0, \mathtt{v}_1)$ of tests, where always $\mathtt{v}_0 = \neg\mathtt{v}_1$. In fact, usually $\mathtt{v}_0 = 0$ and $\mathtt{v}_1 = 1$, corresponding to the values FALSE and TRUE of $\mathtt{v}$, resp.

We formally specify each simple Boolean gate by a set of inequations involving the diamond operator. As an example, consider an OR-gate with inputs in1, in2 and in3 and output out1. To model this gate as an MKA element or, we characterize its behavior by the following inequations:

$$\mathtt{in1}_1 + \mathtt{in2}_1 + \mathtt{in3}_1 \leq |\mathtt{or}\rangle\mathtt{out1}_1 \qquad \mathtt{in1}_0 \cdot \mathtt{in2}_0 \cdot \mathtt{in3}_0 \leq |\mathtt{or}\rangle\mathtt{out1}_0$$

For simplicity, we do not treat negations as blocks but simply swap $\mathtt{v}_0$ and $\mathtt{v}_1$ for a variable v. Every gate gat is deterministic and total; so we require $|\mathtt{gat}\rangle p = |\mathtt{gat}]p$ for every test p. Hence, a quick calculation using shunting shows

$$q \leq |\mathtt{gat}\rangle p \wedge \neg q \leq |\mathtt{gat}\rangle\neg p \Leftrightarrow q = |\mathtt{gat}\rangle p \Leftrightarrow \neg q = |\mathtt{gat}\rangle\neg p$$

This is precisely the shape of the axioms for the OR-gate above.

Moreover, we have to ensure that a block at most modifies its output and internal variables. In the above example, we have to add the inequations $\mathtt{v}_1 \leq |\mathtt{or}\rangle\mathtt{v}_1$ and $\mathtt{v}_0 \leq |\mathtt{or}\rangle\mathtt{v}_0$ for all variables v except out1 as *tracking conditions* for v. The only exception is with the last-evaluated block of an FBD. To allow composition of an FBD with itself and hence also its star iteration, we have to allow that the input channels in the next execution cycle receive new values; so for the last block we drop the above condition for all input variables of the overall FBD. The same holds for output variables which are computed from scratch in every cycle. Note that internal variables have tracking conditions also at the last-evaluated gate because their value is stored for the next cycle. In our formalization, we even omitted the conditions for input variables that are not used later on in the FBD in order to keep the formalization as small as possible.

An input variable that is used in a certain block B but does not appear as input of any block following it (in evaluation order) does not have to be tracked across the program after block B. Its new value will be determined by its input channel in the following execution cycle. For example, in the FBD from the left part of Figure 1, there are neither the inequation $\mathtt{IN4}_0 \leq |\mathtt{or}\rangle\mathtt{IN4}_0$ nor $\mathtt{IN7}_1 \leq |\mathtt{and}\rangle\mathtt{IN7}_1$.

As stated, the overall behavior of a single program cycle can then be described by the product of all blocks in their evaluation order (which is basically a topological sorting corresponding to western reading conventions; for details see [2]). For example, if we consider the whole Figure 1 as a PLC program and denote the blocks by `or1`, `and1` and `sr1`, resp., it corresponds to the expression $\mathtt{or1} \cdot \mathtt{and1} \cdot \mathtt{sr1}$ (recall that negations are modeled by simply swapping $\mathtt{v}_0$ and $\mathtt{v}_1$). If not indicated otherwise, this product describing (a single execution cycle of) an FBD program is named `cycle`; it corresponds to the transition element a from Section 2.2, and we use $\circ$, U, $\Diamond$ and $\Box$ w.r.t. it. It is easy to see that total functionality of the single blocks propagates to the whole program. Following [21], one can give analogous formalizations for the other Boolean gates. In the next section we will deal with the other blocks we introduced in Section 3.1.

3.3 Formalization of Timers

A common mechanism for generating timed signals is shown in Figure 4. There, a TRUE-value on `req` activates the flip-flop (we will discuss its reset input soon) whose output is conjoined with a timer signal of 1 Hz. As long as the output of the flip-flop equals TRUE the counter value will be increased every second by one and is stored in the internal variable `M50`. This behavior will persist as long as `res` does not become TRUE (even if `req` changes its value to FALSE). However, a TRUE-signal of `res` resets both the counter and the flip-flop (note that the flip-flop is reset dominant and that `res` acts as a resetter for both the flip-flop and the counter). The further behavior depends on the value of `req`.

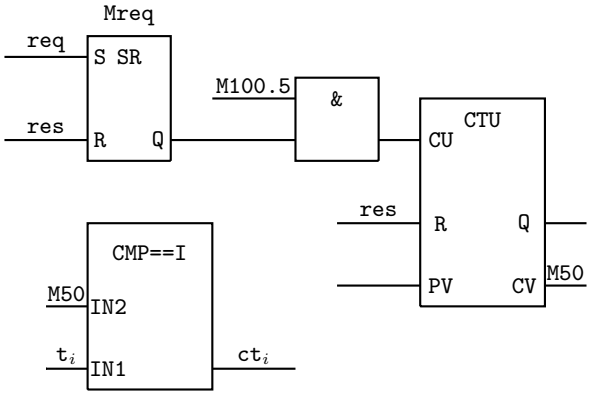

Fig. 4. Generating Time Signals in FBD

For further processing, one often wants to trigger some action at a certain time after starting the counter. In this case, the counter value is compared with

the desired time; the Boolean output is used as trigger signal. This is shown exemplarily in the bottom part of Figure 4: $\mathtt{ct}_i$ becomes TRUE when the output M50 of the counter equals the value of $\mathtt{t}_i$. Usually, for $\mathtt{t}_i$ one uses a constant value to start an action after a given time as we will do in the further course.

In general, one such counter can be associated with several comparators to enable a timed sequence of activations. In this context, one regularly has also a cut-off time tcu after which the timer should be reset to zero. This can be achieved easily by feeding the output of a suitable comparator to the counter's and flip-flop's reset inputs.

In the sequel, we will view the described *timer* component as a black box with inputs req and res as start and reset signals, and the comparator results as outputs. To formalize timers, we arrange the compared values (fed into the IN1 lines of the comparators) in increasing order as a sequence $\mathtt{t}_0$, $\mathtt{t}_1$, ..., $\mathtt{t}_n$, such that the final value $\mathtt{t}_n$ is used as reset input as described above, and denote the (Boolean) outputs of the respective comparators by $\mathtt{ct}_0$, $\mathtt{ct}_1$, ..., $\mathtt{ct}_n$. For better readability, we use in the sequel the notation $\mathtt{ct}_{i,1}$ instead of $(\mathtt{ct}_i)_1$ for indicating a TRUE-value of $\mathtt{ct}_i$ and define $\mathtt{ct}_{i,0}$ analogously. In particular, $\mathtt{ct}_{i,1}$ corresponds to a state where the counter stands at $\mathtt{t_i}$. Then we require the following properties:

- **No simultaneity:** If the output of one timer comparator is TRUE then the others have to be FALSE. In MKA, this can be described by $\mathtt{ct}_{i,1} \leq \prod_{j \neq i} \mathtt{ct}_{j,0}$.
- **Order:** The timer comparators output TRUE according to the above ordering. This means, we have $\mathtt{ct}_{i,1} \leq \Diamond\, \mathtt{ct}_{i+1,1}$ for all $0 \leq i < n$.
- **Resetting:** A TRUE-value of $\mathtt{ct}_n$ resets the counter in the following cycle. Therefore, we have $\mathtt{ct}_{n,1} \leq \bigcirc \mathtt{ct}_{0,1}$.
- **Resting:** We have to ensure that the counter does not start until it gets a request. This means that if the counter stands at zero (modeled by $\mathtt{ct}_{0,1}$) and there is no request then the counter stands at zero also in the subsequent state. To this purpose, we add the requirement $\mathtt{ct}_{0,1} \cdot \mathtt{req}_0 \leq \bigcirc\, \mathtt{ct}_{0,1}$.
- **Starting:** If a resting counter receives a start request it should eventually output $\mathtt{ct}_{1,1}$. This is modeled by the formula $\mathtt{ct}_{0,1} \cdot \mathtt{req}_1 \leq \Diamond\, \mathtt{ct}_{1,1}$.
- **Intermediate states:** The preceding properties deal with situations in which at least the counter comparator outputs TRUE. However, most of the time all the outputs equal FALSE; so we have to deal also with this situation. To ease reading and writing, we introduce the abbreviation $\mathtt{nst} =_{df} \prod_{i=0}^{n} \mathtt{ct}_{i,0}$ (nst stands for 'no significant time'). The mentioned situation occurs if the value of M50 is between two consecutive values of the sequence $\mathtt{t_i}$; so we require $\mathtt{ct}_{i,1} \leq \mathtt{nst}\, \mathsf{U}\, \mathtt{ct}_{i+1,1}$ for all $0 \leq i < n$ (recall the modeling of the until-operator in Section 3.2). Note also that, by the resetting rule, $\mathtt{ct}_{n,1}$ is followed temporally immediately by $\mathtt{ct}_{0,1}$.
- **No other states:** The system is either in a situation where a counter comparator's output is TRUE or it is awaiting another counter comparator's

output to become TRUE. In our framework, this reads

$$\models \mathtt{ct}_{n,1} \vee \bigvee_{i=0}^{n-1} (\mathtt{nst}\,\mathsf{U}\,\mathtt{ct}_{i,1}) \qquad \models \mathtt{ct}_{n,1} + \sum_{i=0}^{n-1} |(\mathtt{nst}\cdot\mathtt{cycle})^*\rangle \mathtt{ct}_{i,1} \quad (4)$$

The counter comparator outputs cannot be altered by any block except the last one in the evaluation ordering. This ensures that time does not change during the execution of one cycle but also may progress between two consecutive cycles. The modeling of this behavior is analogous to that given in Section 3.2.

3.4 A Case Study: Traffic Lights

As an example application we chose an FBD controlling the pedestrian lights of a traffic control signal. With the conventions of Section 3.3 it looks as in Figure 5. If the button is pressed, the pedestrian lights should eventually become green for ten seconds. After a green phase of the pedestrian lights it should take at least nine seconds before the pedestrian lights can become green again (to respect car drivers). Also, there should be a time of three seconds for the car traffic lights to become yellow after pushing the request button. So, if one starts with red pedestrian lights and a timer at zero, a push should lead to green pedestrian lights after three seconds. Then, the lights should stay green for ten seconds, and a new such cycle can start only nine seconds later. Here, the variables have the following definitions and meanings:

- `push` is a Boolean input from the request button of pedestrian lights.
- `gr` is an internal Boolean variable whose value indicates whether the pedestrian lights are green (this value can be forwarded to different output signals; however, for our verification purposes it suffices to consider only `gr` itself).
- `req` and `res` are the start and reset inputs of a timer.
- $\mathtt{c}_0$, $\mathtt{c}_3$, $\mathtt{c}_{13}$ and $\mathtt{c}_{22}$ are the outputs of a timer, corresponding to values of zero, three, thirteen and twenty-two seconds, resp., after starting the timer.

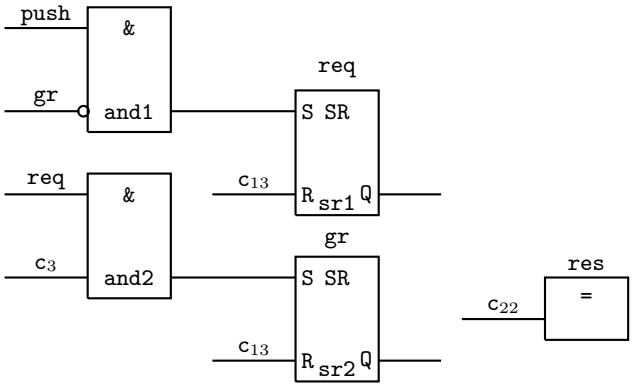

Fig. 5. Pedestrian Lights in Modified FBD

Let us take a short look at the functionality described in Figure 5. A push of the request button has only an effect if the pedestrian lights are not yet green

(this is the purpose of `and1`). If such a push manages to pass `and1` then it activates the timer via the flip-flop `sr1`. The timer start is reset after thirteen seconds (note that this does not stop or reset the timer) to prevent a new activation of the timer after finishing the current cycle. `and2` and `sr2` set the lights to green after three seconds if `req` was set to TRUE by some preceding push of the request button, and reset the lights to red (implicitly by setting `gr` to FALSE) after thirteen seconds. Finally, the timer is reset after twenty-two seconds by the assignment of $\mathtt{c}_{22}$ to the timer reset signal `res`. A sample from the KIV formalization together with explanations can be found in the readme file of [6].

The program should fulfill some temporal properties concerning the interplay between the variable `gr` and the timer. We introduce three examples together with proof sketches in order to give the reader an impression how the rules from Section 2.3 can be used in our context.

- If the timer value is three and there is a request then the lights should be green in the following cycle. The LTL specification of this reads
$$\models \Box\,(\mathtt{req}_1 \wedge \mathtt{c}_{3,1} \rightarrow \circ \mathtt{gr}_1) \tag{5}$$
which can be rewritten in MKA as
$$\models |\mathtt{lig}^*](\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \rightarrow |\mathtt{lig}\rangle \mathtt{gr}_1) \tag{6}$$
Here we used `lig` (short for `lights`) instead of `cycle` as in Section 3.2 for the overall behavior of the system in one cycle. Concretely, we have `lig` = `and1`$\cdot$ `sr1`$\cdot$ `and2`$\cdot$ `sr2` using the namings from Figure 5 (the counter properties are added as axioms to the the formalization, so the assignment of $\mathtt{c}_{22}$ to `res` is covered by the associated `Resetting` rule). One may wonder why we stipulate additionally a request in the precondition. The reason is that it makes later reasoning a lot more convenient, and it is justified by the fact that under reasonable assumptions about the start condition of the system the timer can reach three only if there is an additional request (see also Equations (9)/(10) below).
- Clearly, we are not satisfied with the lights just turning green after three seconds, they should stay green for ten seconds. An LTL formula for this is
$$\models \Box\,(\mathtt{req}_1 \wedge \mathtt{c}_{3,1} \rightarrow \circ(\mathtt{gr}_1 \,\mathsf{U}\, \mathtt{c}_{13,1})) \tag{7}$$
with an equivalent MKA characterization
$$\models |\mathtt{lig}^*](\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \rightarrow |\mathtt{lig}\rangle(|(\mathtt{gr}_1 \cdot \mathtt{lig})^*\rangle \mathtt{c}_{13,1})) \tag{8}$$
- These two properties do not require a certain initial state of the system (the outermost operator is an always operator). However, a start in an inappropriate state can lead to undesired behavior. For example, one can show that $\mathtt{c}_{0,1} \cdot \mathtt{gr}_1 \leq |\mathtt{lig}^*]\mathtt{c}_{0,1} \cdot \mathtt{gr}_1$ holds, which means that the lights stay green all the time. This initial state contradicts the intention of the program, because the lights should turn green only after the timer reaches the value three, and they have to be set to red again before the timer is reset; so the state $\mathtt{c}_{0,1} \cdot \mathtt{gr}_1$ would represent an inconsistency. A reasonable choice for the initial state is that the lights are red and the timer is zero. Starting in such a state and pushing the button while the lights are red, we want that from the next state we eventually reach a state with the following properties:

- the lights are green,
- the counter value is thirteen, and
- in the next state the lights are red until the counter reaches 22.

Note that this is essentially a statement only about the last cycle where the timer value is thirteen. We chose this example because it is well suited as an illustration of the application of our techniques. An LTL-formulation of this property is

$$\models \mathtt{gr}_0 \wedge \mathtt{c}_{0,1} \rightarrow \Box\,(\mathtt{gr}_0 \wedge \mathtt{push}_1 \rightarrow \circ(\Diamond\,\mathtt{gr}_1 \wedge \mathtt{c}_{13,1} \wedge \circ(\mathtt{gr}_0\,\mathsf{U}\,\mathtt{c}_{22,1}))) \quad (9)$$

and its translation into MKA reads

$$\begin{array}{l} \models \mathtt{gr}_0 \cdot \mathtt{c}_{0,1} \rightarrow \\ \quad |\mathtt{lig}^*](\mathtt{gr}_0 \cdot \mathtt{push}_1 \rightarrow \\ \qquad\quad |\mathtt{lig}\rangle(|\mathtt{lig}^*\rangle\mathtt{gr}_1 \cdot \mathtt{c}_{13,1} \cdot |\mathtt{lig}\rangle|(\mathtt{gr}_0 \cdot \mathtt{lig})^*\rangle\mathtt{c}_{22,1})) \end{array} \quad (10)$$

Property (6) is rather easy to prove: First, we note that by (3) it suffices to show $\models \mathtt{req}_1 \cdot \mathtt{c}_{3,1} \rightarrow |\mathtt{lig}\rangle\mathtt{gr}_1$ due to the property $|x]1 = 1$ in all MKAs. Equivalently, we can show that $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle\mathtt{gr}_1$ holds. This is done easily by unfolding the definition of `lig` and iterated application of modality and isotony of the diamond with the aid of the characterization of the respective blocks.

Similarly, we can prove **Property** (8) by showing the inequation $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle|(\mathtt{gr}_1 \cdot \mathtt{lig})^*\rangle\mathtt{c}_{13,1}$ which is an example for the application of Lemma 3. This means we have to show $\mathtt{gr}_1 \cdot \neg\mathtt{c}_{13,1} \leq |\mathtt{lig}\rangle\mathtt{gr}_1$ and $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle(\mathtt{gr}_1 \cdot |\mathtt{lig}^*\rangle\mathtt{c}_{13,1})$. The first inequation can be shown analogously to the proof sketch of Equation (6). Due to total functionality, definition of the diamond and distributivity, the second one can be split up into $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle\mathtt{gr}_1$ and $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle|\mathtt{lig}^*\rangle\mathtt{c}_{13,1}$. The first inequation is already known from the proof of Equation (6). For the last one, we have the chain of inequalities $\mathtt{req}_1 \cdot \mathtt{c}_{3,1} \leq \mathtt{c}_{3,1} \leq |\mathtt{lig}\rangle|(\mathtt{nst}\cdot\mathtt{lig})^*\rangle\mathtt{c}_{13,1} \leq |\mathtt{lig}\rangle|\mathtt{lig}^*\rangle\mathtt{c}_{13,1}$ by isotony, timer properties and isotony of multiplication and diamond.

For **Property** (10) we will resort only to rough explanations and refer the reader for details to the full KIV project file [6]. As in the previous cases, we transform the claim into the equivalent inequation

$$\mathtt{gr}_0 \cdot \mathtt{c0}_1 \leq |\mathtt{lig}^*](\mathtt{gr}_0 \cdot \mathtt{push}_1 \rightarrow (|\mathtt{lig}^*\rangle\mathtt{gr}_1 \cdot \mathtt{c}_{13,1} \cdot |\mathtt{lig}\rangle|(\mathtt{gr}_0 \cdot \mathtt{lig})^*\rangle\mathtt{c}_{22,1}))$$

Here we can simplify the right side using the equality $\mathtt{gr}_1 \cdot \mathtt{c}_{13,1} = \mathtt{gr}_1 \cdot \mathtt{c}_{13,1} \cdot |\mathtt{lig}\rangle|(\mathtt{gr}_0 \cdot \mathtt{lig})^*\rangle\mathtt{c}_{22,1}$ (this follows from $\mathtt{gr}_1 \cdot \mathtt{c}_{13,1} \leq |\mathtt{lig}\rangle|(\mathtt{gr}_0 \cdot \mathtt{lig})^*\rangle\mathtt{c}_{22,1}$ which in turn can be shown using timer properties and Lemma 3) and reduce our task — after exploiting isotony — to showing $\mathtt{gr}_0 \cdot \mathtt{c}_{0,1} \leq |\mathtt{lig}^*](\mathtt{gr}_0 \cdot \mathtt{push}_1 \rightarrow (|\mathtt{lig}^*\rangle\mathtt{gr}_1 \cdot \mathtt{c}_{13,1}))$. Introducing the abbreviations $\mathtt{init} =_{df} \mathtt{gr}_0 \cdot \mathtt{c}_{0,1}$ and $\mathtt{result} =_{df} |\mathtt{lig}^*\rangle\mathtt{gr}_1 \cdot \mathtt{c}_{13,1}$ this goal reads $\mathtt{init} \leq |\mathtt{lig}^*](\mathtt{gr}_0 \cdot \mathtt{push}_1 \rightarrow \mathtt{result})$. Now we use the timer's no-other-state property (4) and replace the above inequation by seven of the type $\mathtt{init} \leq |\mathtt{lig}^*](\mathtt{gr}_0 \cdot \mathtt{push}_1 \cdot \mathtt{itm} \rightarrow \mathtt{result})$ where the intermediate value `itm` has the form $\mathtt{itm} = \mathtt{c}_{\mathtt{i},1}$ for $\mathtt{i} = 0, 3, 13, 22$ or $\mathtt{itm} = |(\mathtt{nst} \cdot \mathtt{lig})^*\rangle\mathtt{c}_{\mathtt{j},1}$ with $\mathtt{j} = 3, 13, 22$. Some of these cases can be handled by showing that the argument of the diamond evaluates to 1. For the remaining properties, a crucial point is the application of Lemmata 1 and 2 together with appropriate timer properties, isotony and MKA calculus.

Summing up our experiences, we can say that after some time of familiarization, proving in KIV became routine work without greater difficulties. An increasing amount of calculation rules and lemmata in MKA made it a pleasant task.

4 Conclusion and Outlook

After proving some useful LTL rules concerning the until-operator in MKA we applied them successfully to a considerable extension of the verification framework developed in [21].

Now that the theoretical foundations are laid, notably concerning the treatment of timing issues, the next step will be to tackle the verification of larger, more lifelike systems. A great help for this goal will be be the by now substantial body of reusable rules and lemmata we have accumulated. Other topics of future work concern the automated construction of input files, comparison with model checkers and extension of the approach to other PLC languages.

Another point is a formal proof of the properties from Section 3.3. In the present paper, these rules were inserted as axioms without further verification. A proof needs meta-knowledge about the natural numbers which has to be added in some way. Moreover, one has to assume and to model the condition that the execution time of one cycle of the PLC does not exceed the period of the used frequency generator. Otherwise, effects similar to the Nyquist-Shannon sampling theorem (see [44]) would destroy the functionality of Figure 4.

Acknowledgement We are grateful to the anonymous referees for their careful scrutiny and helpful remarks.

References

1. Coq. `https://coq.inria.fr/`. [Online; accessed 7-July-2015].
2. IEC61131. `http://webstore.iec.ch/webstore/webstore.nsf/artnum/048541!opendocument`. [Online; accessed 20-March-2018].
3. The KIV system. `http://www.isse.uni-augsburg.de/en/software/kiv/`. [Online; accessed 20-March-2018].
4. NuSMVExamples. `http://nusmv.fbk.eu/examples/examples.html`. [Online; accessed 7-August-2018].
5. Step7. `http://w3.siemens.com/mcms/simatic-controller-software/en/step7/Pages/Default.aspx`. [Online; accessed 20-March-2018].
6. Verification of pedestrian lights in MKA. `http://rolandglueck.de/Downloads/Pedestrian_lights_verified.zip`. [Online; accessed 20-March-2018].
7. VerifyThis 2015. `http://verifythis2015.cost-ic0701.org/results`. [Online; accessed 8-August-2018].
8. VerifyThis 2017. `http://www.pm.inf.ethz.ch/research/verifythis/Archive/2017.html`. [Online; accessed 8-August-2018].
9. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
10. R.-J. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

11. M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 3rd. edition, 2012.
12. R. Berghammer, I. Stucke, and M. Winter. Using relation-algebraic means and tool support for investigating and computing bipartitions. *J. Log. Algebr. Meth. Program.*, 90:102–124, 2017.
13. G. Birkhoff. *Lattice Theory*. Amer. Math. Soc., 3rd edition, 1967.
14. P. Brunet, D. Pous, and I. Stucke. Cardinalities of Finite Relations in Coq. In J. C. Blanchette and S. Merz, editors, *ITP 2016, Proceedings*, volume 9807 of *LNCS*, pages 466–474. Springer, 2016.
15. H. Carlsson, B. Svensson, F. Danielson, and B. Lennartson. Methods for Reliable Simulation-Based PLC Code Verification. *IEEE Trans. Industrial Informatics*, 8(2):267–278, 2012.
16. J. Desharnais and B. Möller. Non-associative Kleene Algebra and Temporal Logics. In P. Höfner, D. Pous, and G. Struth, editors, *RAMiCS 2017, Proceedings*, volume 10226 of *LNCS*, pages 93–108, 2017.
17. J. Desharnais, B. Möller, and G. Struth. Modal Kleene algebra and applications - a survey. *Journal on Relational Methods in Computer Science*, 1:93–131, 2004.
18. T. Ehm, B. Möller, and G. Struth. Kleene modules. In R. Berghammer, B. Möller, and G. Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science*, volume 3051 of *LNCS*, pages 112–124. Springer, 2004.
19. J. Ertel. Verifikation von SPS-Programmen mit Kleene Algebra. Master's thesis, Institut of Informatics, University of Augsburg, 2017.
20. Z. Ésik, U. Fahrenberg, A. Legay, and K. Quaas. Kleene Algebras and Semimodules for Energy Problems. In D. V. Hung and M. Ogawa, editors, *ATVA 2013, Proceedings*, volume 8172 of *LNCS*, pages 102–117. Springer, 2013.
21. R. Glück and F. B. Krebs. Towards Interactive Verification of Programmable Logic Controllers Using Modal Kleene Algebra and KIV. In W. Kahl, M. Winter, and J. N. Oliveira, editors, *RAMiCS 2015, Proceedings*, volume 9348 of *LNCS*, pages 241–256. Springer, 2015.
22. M. Gondran and M. Minoux. *Graphs, Dioids and Semirings*. Springer, 2008.
23. W. Guttmann. Stone Relation Algebras. In P. Höfner, D. Pous, and G. Struth, editors, *RAMiCS 2017, Proceedings*, volume 10226 of *LNCS*, pages 127–143, 2017.
24. P. Höfner and B. Möller. Dijkstra, Floyd and Warshall meet Kleene. *Formal Asp. Comput.*, 24(4-6):459–476, 2012.
25. M. Hollenberg. An equational axiomatization of dynamic negation and relational composition. *J. Logic, Language and Information*, 6(4):381–401, 1997.
26. M. Hollenberg. Equational axioms of test algebra. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, CSL '97*, volume 1414 of *LNCS*, pages 295–310. Springer, 1997.
27. M. Jackson and R. McKenzie. Interpreting Graph Colorability in Finite Semigroups. *IJAC*, 16(1):119–140, 2006.
28. E. Jee, J. Yoo, S. D. Cha, and D.-H. Bae. A data flow-based structural testing technique for FBD programs. *Information & Software Technology*, 51(7):1131–1139, 2009.
29. P. Jipsen and H. Rose. *Varieties of Lattices*. Springer, 1st edition, 1992.
30. W. Kahl. Graph Transformation with Symbolic Attributes via Monadic Coalgebra Homomorphisms. *ECEASST*, 71, 2014.
31. Y. Kawahara and H. Furusawa. An algebraic formalization of fuzzy relations. *Fuzzy Sets and Systems*, 101(1):125–135, 1999.
32. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

33. D. Kozen. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.
34. F. Kröger and S. Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
35. J. Li, A. Qeriqi, M. Steffen, and I. C. Yu. Automatic translation from FBD-PLC-programs to NuSMV for model checking safety-critical control systems. In *NIK 2016*. Bibsys Open Journal Systems, Norway, 2016.
36. T. Litak, S. Mikulás, and J. Hidders. Relational Lattices. In P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, editors, *RAMiCS 2014, Proceedings*, volume 8428 of *LNCS*, pages 327–343. Springer, 2014.
37. E. Manes and D. Benson. The Inverse Semigroup of a Sum-ordered Semiring. *Semigroup Forum*, 31:129–152, 1985.
38. G. Michels, S. J. C. Joosten, J. van der Woude, and S. Joosten. Ampersand - Applying Relation Algebra in Practice. In H. C. M. de Swart, editor, *RAMICS 2011, Proceedings*, volume 6663 of *LNCS*, pages 280–293. Springer, 2011.
39. B. Möller, P. Höfner, and G. Struth. Quantales and Temporal Logics. In M. Johnson and V. Vene, editors, *AMAST 2006, Proceedings*, volume 4019 of *LNCS*, pages 263–277. Springer, 2006.
40. B. Möller and P. Roocks. An algebra of database preferences. *J. Log. Algebr. Meth. Program.*, 84(3):456–481, 2015.
41. J. N. Oliveira. A relation-algebraic approach to the "Hoare logic" of functional dependencies. *J. Log. Algebr. Meth. Program.*, 83(2):249–262, 2014.
42. O. Pavlovic and H.-D. Ehrich. Model Checking PLC Software Written in Function Block Diagram. In *ICST 2010*, CEUR Workshop Proceedings. IEEE Computer Society, 2010.
43. V. Pratt. Dynamic algebras: Examples, constructions, applications. *Studia Logica*, 50:571–605, 1991.
44. C. E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10 – 21, 1949.
45. K. Solin and J. von Wright. Enabledness and termination in refinement algebra. *Sci. Comput. Program.*, 74(8):654–668, 2009.
46. B. von Karger. Temporal algebra. *Mathematical Structures in Computer Science*, 8(3):277–320, 1998.
47. H. Wan, G. Chen, X. Song, and M. Gu. Formalization and verification of PLC timers in Coq. In S. I. Ahamed, E. Bertino, C. K. Chang, V. Getov, L. Liu, H. Ming, and R. Subramanyan, editors, *COMPSAC 2009, proceedings*, pages 315–323. IEEE Computer Society, 2009.
48. S. Wimmer and P. Lammich. Verified model checking of timed automata. In D. Beyer and M. Huisman, editors, *ETAPS 2018, Proceedings, Part I*, volume 10805 of *LNCS*, pages 61–78. Springer, 2018.